C4

```
 1
 2  /*
 3  ** ************************************************************
 4  **
 5  ** File Name:   RSLstart.c
 6  **
 7  ** Copyright (c) 1998,1999 by EMC Corporation.
 8  **
 9  ** Purpose:
10  ** --------
11  **     The intent of the contents of this file is to implement the
12  **     functions the control execution of the restore for the
13  **     Restore Service
14  **     Library.
15  **
16  **     These functions are provided to allow:
17  **     - creation of submit objects
18  **          which define the set of objects to be
19  **     restored and the scripts to be run before and a restoration.
20  **     - starting the restoral of a submit object.
21
22  **     The following functions comprise restoral management:
23
24  **     RSTSL_Start
25
26  ** Compile-Time Options:
27  **     This section must list any compile time definitions
28  **     which will affect this header.
29  **
30  ** ************************************************************
31  ** Feature test switches.
32  ** Standard defines required to turn on OS features go here.
33  **
34  ** The following is required for code that uses POSIX API's.
35  ** Remove for non-POSIX, non-portable code.
36  */
37  #define _POSIX_SOURCE 1
41  /*
42  ** System headers.
43  */
45  #include <sys/wait.h>
47  /*
48  ** Epoch headers.
49  */
50  #include <eb/eb_port.h>
51  #include <eb/rb_log.h>
52  #include <eb/utili/eb_normalize.h>
53  #include <eb/utili/ebutil.h>
54  #include <ebreport/abvil.h>
56  /*
57  ** Local headers
58  */
59
```

```
 62  #include <RSLinterns.h>
 63  #include <RSLauxSupp.h>
 64  #include <restore/EDMRRSubmitiapi.h>
 66  #include <restore/RBprogmsg.h>
 67  #include <restore/dispatch_daemon.h>
 68  #include <restore/EDMRProgressapi.h>
 70  extern int RunExecutable(const boolean_ty ResetUid,
 71                           const int RunUid,
 72                           const char *starting_cwd,
 73                           char *executable_name,
 74                           char *executable_argv,
 75                           int *run_env,
 76                           int *run_exit_status,
 77                           boolean_ty *run_cancelled,
 78                           boolean_ty (*QuitTest) (void));
 80  extern int RunWorkItemRestores(int SubmitObjectID, boolean_ty (*CancelRestoreTest) ());
 82  static errno_ty
 83  ExecuteWorkItemRestore(int SubmitObjectID,
 84                         boolean_ty (*QuitTest)(void));
 86  static errno_ty
 87  RunPrepareRestore(int SubmitObjectID,
 88                    boolean_ty (*QuitTest)(void),
 89                    int *PrepareExit);
 91  static errno_ty
 92  RunCleanupRestore(int SubmitObjectID,
 93                    boolean_ty (*QuitTest)(void),
 94                    int runphase_status,
 95                    int *CleanupExit);
 96  /* #defines, structures, typedefs local to this source file
 98
 99  #define STR_SUBR(str) (str)\
100      int rem_nl_index;\
101      for(rem_nl_index = 0; str[rem_nl_index] != '\0'; rem_nl_index++)\
102      {\
103          if(str[rem_nl_index] == '\n')\
104              str[rem_nl_index] = '\0';\
105      }\
106  }
107  #define REMOVE_NEWLINE(str)\
108  {\
109
110  /*
111  * Start
112  *
113  * This function begins execution of the restoral of the objects in a
114  * submit object. Its progress is reported and requests for operator input are
115  * returned via RSTSL_GetRestoreFeedback.
116  *
117  * Parameters:
118  *
119  * SubmitObjectID (I) - ID of the submit object which describes the restore
120  *
121  * QuitTest (I) - function to call to check for quit signal
122  */
123  errno_ty RSTSL_Start( int SubmitObjectID, boolean_ty (*QuitTest) (
124                        void))
125  {
```

```
127  2      int  ret_pre;
130  2      int  ret_exec;
131  2      int  ret_post;
132  2      int  ret_all_ok;
133  2      int  PrepareExit = 0;
133  2      int  CleanupExit = 0;
133  2      boolean_ty QuitFlag = FALSE;
135  2
135  1      char  sactime[32];
137  1
137  1      memset(sactime, 0, 32);
139  1
139  1      (void)time_r(&rcp->rc_cmd_starttime);
141  1      (void)ctime_r(&rcp->rc_cmd_starttime, sactime, 32);
143  1      rcp->rc_cmd_last_waf_time = rcp->rc_cmd_starttime;
145  1
145  1      REMOVE_NEWLINE(sactime);
147  1
147  1      rbe_log_stats(0, "Restore Started at %s.", sactime);
149  1      rbe_log_stats(0, "Restore Started application type %s.",
150  1                       (rcp->rc_backup_app == 0)
151  1                     ? "Network"
152  1                     : (struct pluginData)
153  1                       rcp-> currentPlgt->
                                            idData)
                                            name );
156  1      rbe_log_stats(0, "Restore Started of ",
                       template %s, trailset %s.",
157  1                       STR_SURE(rcp-> top_level_object),
158  1                       STR_SURE(rcp->rc_template_name);
159  1                       STR_SURE(rcp-> rc_saveset_thread) ? "Alternate" : "Primary");
162  1      rbe_log_stats(0, "Restore Started by user %s, Uid %d, Gid %d.",
163  1                       STR_SURE(rcp-> rc_human_uidname),
164  1                       rcp->rc_human_uid, rcp->rc_human_gids[0]);
166  1      rbe_log_stats(0, "Restore Started with client destination %s.",
167  1                       STR_SURE(rcp->rc_client_hostname));
169  1      /* if not a network restore,
169  1                   check if plugin has its own start function */
171  1      if ( rcp-> rc_backup_app != 0
172  1        && NULL != rcp-> currentPlgt-> pifuncArray)
173  2      {
174  2          ret_exec = rcp-> currentPlgt-> pifuncArray[
174  2                      PIFuncIndexStartRestore ]
175  2                      /* set RE's internal status */
176  2          ret_exec = rcp-> currentPlgt-> pifuncArray[
176  2                      PIFuncIndexStartRestore ]
178  2                      rcp. SubmitObjectID, QuitTest);
179  2
179  2      if ( QuitTest() )     /* check for abort before return */
179  3      {
180  3          rcp. SubmitObjectID, QuitTest);
181  3          rbe_log_stats( EP_RB_RECOVER_ABORT,
181  3                      "The restore was quit by the user during
182  3                      execution." );
182  3          setGlobalStatus( EMKR_STATE_USER_QUIT );  /* set RE's internal status */
```

```
183  1      }
184  1
184  1          return EP_RB_RECOVER_ABORT;
186  2
186  2      if (E_SUCCESS != ret_exec)
187  2          setGlobalStatus( EMKR_STATE_FAILED );      /* set RE's internal status */
189  2      else
189  2          setGlobalStatus( EMKR_STATE_SUCCESSFUL );  /* set RE's internal status */
191  2
191  1          return( ret_exec );
192  1      }
183  1      ret_pre = RunPrepareRestore(SubmitObjectID,
                                      QuitTest,
                                      &PrepareExit);
202  1      }
203  2      if(PrepareExit != 0)
204  1      {
204  1          rbe_log_stats(EP_RB_RECOVER_PREFAILED,
206  1                      "The restore failed during preparation. Exit %d"
207  1                      PrepareExit);
208  1          setGlobalStatus( EMKR_STATE_FAILED );  /* set RE's internal status */
209  1          return (EP_RB_RECOVER_PREFAILED);
211  2          ret_exec = ExecuteWorkItemRestore(SubmitObjectID,
213  2                      QuitTest);
213  1      }
215  1      if (E_SUCCESS == ret_exec)   /* check for abort before cleanup */
216  1      {
217  2          QuitFlag = QuitTest();
218  1      }
220  1      if( QuitFlag )
221  1      {
223  1
224  2      int  local_stat;
225  2      EDMStats stats;
227  2      memset( &stats, 0, sizeof(EDMStats) );
229  2      if (0 == getRestoreStatus( 0, &stats, &local_stat ))
230  2      {
231  2          ret_local_stat;
232  2          ret_all_ok = ret_all_ok   /* Internal error: Failed in getRestoreStatus */
234  2      }
235  2      else if (stats.edm_failed)    /* if any worItems failed */
236  1      {
236  1          if (0 == stats.edm_successful)
237  1              ret_all_ok = EP_RB_RECOVER_ALLFAIL;
238  1          else if (stats.edm_successful > stats.edm_failed)
239  1              ret_all_ok = EP_RB_RECOVER_FEWFAIL;
```

```
240 3            else
241 3                ret_all_ok = EP_RB_RECOVER_MANYFAIL;
242 2
243 1        }
244 1
245 1        ret_post = RunCleanupRestore(SubmitObjectID,
246 1                                     QuitTest,
247 1                                     ret_all_ok,
248 1                                     &CleanupExit);
249 1
250 1        if(QuitFlag)
251 2        {
252 2            rbe_log_stats(EP_RB_RECOVER_ABORT,
253 2                          "The restore was quit by the user during execution.",
254 2                          ");
255 1            return EP_RB_RECOVER_ABORT;       /* set RE's internal status */
256 1        }
257 1
258 1        if (E_SUCCESS != ret_exec)            /* return execute status if it failed */
259 2            setGlobalStatus( EMRE_STATE_FAILED );   /* set RE's internal status */
260 2
261 1        return ret_exec;
262 1
263 1        if (EP_RB_RECOVER_ABORT == ret_post)
264 2            rbe_log_stats(EP_RB_RECOVER_ABORT,
265 2                          "The restore was quit by the user during cleanup.");
266 2            setGlobalStatus( EMRE_STATE_USER_QUIT );   /* set RE's internal status */
267 2
268 2            return EP_RB_RECOVER_ABORT;        /* set RE's internal status */
269 2
270 1        }
271
272 2        if( (CleanupExit != 0) || (E_SUCCESS != ret_post) )
273 2        {
274 2            rbe_log_stats(EP_RB_RECOVER_ABORT,
275 2                          "The restore failed during cleanup. Exit %d",
276 2                          CleanupExit);
277
278 2            setGlobalStatus( EMRE_STATE_FAILED );      /* set RE's internal status */
279 2            return (EP_RB_RECOVER_POSTFAILED);        /* set RE's internal status */
280 1        }
281 1
283 1        setGlobalStatus( EMRE_STATE_SUCCESSFUL );      /* set RE's internal status */
285 1        return( E_SUCCESS );
287                                                        /* RSTSL_Start */
          }
```

```
292     static eerror_ty
293     ExecuteWorkItemRestore(int SubmitObjectID,
294                            boolean_ty (*QuitTest)(void))
297 {
298         int ret_RunWItem;
299         sm_push();
301 1       rcp->error_message[0] = 0;
303 1       if(0 != (ret_RunWItem = RunWorkItemRestore(
304 2                                  SubmitObjectID, QuitTest)))
305 2           rbe_log_stats(0, "internal error: Failed in RunWorkItemRestore");
306 2
308 1       sm_pop();
310 1       if (QuitTest() == TRUE)
311 1           return EP_RB_RECOVER_ABORT;
313 1       if (ret_RunWItem != 0)
314 1           return EP_RB_RECOVER_EXECFAILED;
316 1       return E_SUCCESS;
317     }
```

```c
320   #define EXECUTABLE_MAX 1024
321   static errno_ty
322   static errno_ty
323   RunPrepareRestore(int SubmitObjectID,
324                     boolean_ty (*QuitTest)(void),
325                     int *PrepareExit)
326   {
327       char **prephaseargs = NULL;
328 1     char **prephaseenv = NULL;
329 1     int GetSOStatus = 0;
330 1     char preExecutable[EXECUTABLE_MAX];
331       boolean_ty restore_cancelled = FALSE;
332 1     *PrepareExit = 0;
334       /*
335        * GetSOPrePhase allocates prephaseargs & prephaseenv.
336        * This will need to be free'ed later.
337        */
338 1
340 1     if(0 != GetSOPrePhase(SubmitObjectID,
341                             preExecutable,
342                             EXECUTABLE_MAX,
343                             &prephaseargs,
344                             &prephaseenv,
345                             &GetSOStatus))
346 2     {
347         rbe_log_stats(0,"Internal error: Failed in GetSOPrePhase")
348 2        return (EP_RB_RECOVER_FATALERR);
349 1     }
351 1     if(0 != strcmp(preExecutable, ""))
352 2     {
353 2        setGlobalStatus( EDMRE_STATE_PREPHASE );
354 2        if(-1 == RunExecutable(FALSE,
355                                  0,
356                                  NULL,
357                                  preExecutable,
358                                  prephaseargs,
359                                  prephaseenv,
360                                  PrepareExit,
361                                  &restore_cancelled,
362                                  QuitTest))
363 2        {
364 3           rbe_log_stats(
365 3              0,"Internal error: Failed in RunExecutable for prepare.");
366 3           return (EP_RB_RECOVER_FATALERR);
367 2        }
368 2        if(TRUE == restore_cancelled)
369 2           return (EP_RB_RECOVER_ABORT);
370 1     }
372 1     return( E_SUCCESS );
373   }
```

```c
375   boolean_ty alwaysFalse() { return FALSE; }
```

```c
373  1  static errno_t
378     RunCleanupRestore (int SubmitObjectID,
379                        boolean_t (*QuitTest) (void),
380                        int runphase_status,
381                        int CleanupExit)
382  1  {
384  1     char **postphaseargs = NULL;
185  1     char **postphaseenv = NULL;
186  1     int GetSOStatus = 0;
187  1     boolean_t postExecutable=EXECUTABLE_MAX;
188  1     boolean_t restore_cancelled = FALSE;
189  1     boolean_t ignore_quit=FALSE;

391  1     *CleanupExit = 0;

           /*
            * GetSOPostPhase allocates postphaseargs & postphaseenv.
            * This will need to be free'd later.
            */
395  1     PurgeTrailQueue();
396  1     if(0 != GetSOPostPhase(SubmitObjectID,
397                              postExecutable,
398                              EXECUTABLE_MAX,
399                              &postphaseargs,
400                              &postphaseenv,
401                              &GetSOStatus))
402         {
403            the_log_stats(0,"Internal error: Failed in GetSOPostPhase")
404            return (EP_RB_RECOVER_FATALIER);
405         }

407  2  #define RESTORE_BREAK       "RESTORE_BREAK="
408  2  #define RESTORE_BREAK_TRUE  "RESTORE_BREAK=T"
409     #define RESTORE_BREAK_ERROR "RESTORE_BREAK=E"

410  1  if(0 != strcmp(postExecutable, ""))
411  2  {
412  2     /*
413  2      * If a quit has been specified, we need to tweak the
414  2      * RESTORE_BREAK environment variable if set
415  2      */
416  2     char *abort=NULL;
417  2     if(QuitTest())
418  3     {
419  3        abort=RESTORE_BREAK_TRUE;
420  2     }
421  2     else if(0!=runphase_status)
422  3     {
423  3        abort=RESTORE_BREAK_ERROR;
424  2     }

           /*
            * ignore_quit is set to 1 when we have already processed a BREAK
            * (CANCEL) from gui, and are using the environment variable
            * RESTORE_BREAK_B to signal to the post restore script to clean
            * up form this break. When that happens, an ignore_quit value
            * of 1 will cause actual quit signals to be ignores by the
            * cleanup script, since we already know (via the environment
            * variable) that we are in 'cleanup mode' and no further signal
            * interception is necessary.
            */
430  2     ignore_quit=FALSE;
431  2     if (NULL!=runphase_statusv)
            if (NULL!=postphaseenv)
```

```c
442  3        int isub=0;
443  3        char *cptr;
444  3        while(cptr=postphaseenv[isub])
445  4        {
446  4           if(strcmp(postphaseenv[isub], _resl_stdrdup(abort))==0)
447  5           {
448  5              postphaseenv[isub]=_resl_stdrdup(abort);
449  5              ignore_quit=TRUE;
450  5              if(NULL==postphaseenv[isub])
451  6              {
452  6                 the_log_stats(
453  6                    EP_RB_RECOVER_MALLOC_FAILURE,"Allocate failed in RS.start.c");
454  6                 return EP_RB_RECOVER_POSTFAILED;
455  5              }
456  5           }
457  4           isub++;
458  4        }
459  3     }
460        }

461  2     setGlobalStatus( EDMRE_STATE_POSTPHASE );    /* set RE's internal status */
462  2     if(-1 == RunExecutable(FALSE,
463  3                            0,
464  3                            NULL,
465  3                            postExecutable,
466  3                            postphaseargs,
467  3                            postphaseenv,
468  3                            CleanupExit,
469  3                            restore_cancelled,
470  3                            ignore_quit?alwaysFalse:QuitTest))
471  2     {
472  2        the_log_stats(
473  3           0,"Internal error: Failed in RunExecutable for cleanup.");
474  3        return (EP_RB_RECOVER_FATALIER);
475  2     }
476  2     if(TRUE == restore_cancelled)
477  3        return (EP_RB_RECOVER_ABORT);
478  1  }
479  1  return( E_SUCCESS );
480     }
```

```
482    static eerrno_ty
483    RunExecutionOverrideRestore(int SubmitObjectID,
484                                boolean_ty (*QuitTest)(void))
485    {
486
487 1      return( E_SUCCESS );
488    }
```

```
489    #undef EXECUTABLE_MAX
```

```
  1
  2   /* ********************************************************************
  3    **
  4    **  File Name:    RSLwlsvr.c
  5    **
  6    **  Copyright (c) 1998,1999 by EMC Corporation.
  7    **
  8    **  Purpose:
  9    **  -------
 10    **  The intent of the contents of this file is to implement the
 11    **  functions that control execution of work item restores for
 12    **  Restore
 13    **  Service Library.
 14    **
 15    **  The following functions comprise restoral management:
 16    **
 17    **      RunWorkItemRestores()
 18    **
 19    **  Compile-Time Options:
 20    **  -------
 21    **      This section must list any compile time definitions
 22    **      which will affect this header.
 23    **
 24    **  These functions are provided to allow:
 25    **
     *****/
 27   #define _POSIX_SOURCE 1
 30   /*
 31    ** System headers.
 32    */
 34   #include <sys/time.h>
 35   #include <sys/types.h>
 36   #include <sys/wait.h>
 37   #include <values.h>
 39   /*
 40    ** Epoch headers.
 41    */
 42   #include <eb/eb_port.h>
 43   #include <eb/rb_log.h>
 44   #include <eutil/eutil.h>
 45   #include <restore/hbprogimg.h>
 47   /*
 48    ** Local headers.
 49    */
 51   #include <RSLapexrates.h>
 52   #include <RSLremd.h>
 53   #include <EDMRESchedApi.h>
 54   #include <EDMRESubmitApi.h>
 55   #include <RSListerms.h>
 56   #include <RSLrbrmain.h>
 57   #include <restore/EDMRESubmitApi.h>
 58   #include <EDMREDrvAPI.h>
 60   #define STR_SURE(str)  (str) ? str:""
 63   /*
 66   /*
```

```
 64    *
 65    **  RunWorkItemRestores
 66    *
 68    **  Set the number of drives being used for the life of the restore.
 69    **  SetQuitFlag = FALSE ( # of trail restores )
 70    **  TrailRestoresLeft = ( # of trail restores )
 71    **  TrailRestoresRunning = 0;
 72    *
 73    **  while drive available.
 74    **      RunTrail -- Set drive concurrency for trail restore.
 75    **      TrailRestoresRunning++;
 76    **      while OKToRunWIForTrail
 77    **          StartWIRestore
 78    *
 79    **  while(1)
 80    **      if(QuitTest)
 81    **          Send WICancells
 82    **          SetQuitFlag = TRUE
 83    **      Select(from pipe, timeout (5 seconds))
 84    **      for each WI that completes
 85    **          interprate return.
 86    *
 87    **          Drain progress.
 88    **          Seek in progress for work item.
 89    **          if(OKToReschedule && SetQuitFlag == FALSE)
 90    **              Add to TrailQueue
 91    **          if(TrailRestoreHasMoreWorkitems && SetQuitFlag == FALSE)
 92    **              while OKToRunWIForTrail
 93    **                  StartWIRestore
 94    **          else  -- RunNewTrailRestore
 95    **              EndTrailRestore(prevTrail)Queue
 96    **              TrailRestoresLeft--;
 97    **              TrailRestoresLeft < 0
 98    **              while (drive available &&
 99    **                     TrailRestoresLeft)
100    *
101    **                  RunTrail -- Set drive concurrency for trail restore.
102    **                  TrailRestoresRunning++;
103    **                  while (
104    **                    OKToRunWIForTrail && SetQuitFlag == FALSE)
105    **                      StartWIRestore
106    **                  end while
107    **              end else
108    **          End for each WI completes
109    **          if ((!SetQuitFlag == TRUE) && (TrailRestoresLeft == 0))||
110    **                (TrailRestoresLeft == 0))
111    **              return;
112    **      end exit loop.
113    *
114    **  end while(1)
115    */
118   /* Stubs */
120   static int InterpretWorkItemRestoreResults(wi_restore_results *results);
122   static int test_fdinit (fd);
125   static void DebugLogFds(char *error_msg;
127                           fd_set *fds);
```

```c
129  static int
130      DetermineGlobalDrivesUse();

132  static int
133      test_fd_hup(int fd);

135  static int
136      FindTrailIDForWItem(int handle,
137                          int TrailID,
138                          int *status);

140  static int
141      SendRunningWorkItemsQuit();

     /* End Stubs */

143  static int
144      Select(nfds,
145             fd_set *readfds,
146             fd_set *writefds,
147             fd_set *exceptfds,
148             struct timeval *timeout);
149

150  static int
153      RunWorkItemRestoreResults(const int TrailID,
154                                int *TrailID,
155                                wi_restore_results *results);

158  /*
159   * RunWorkItemRestores()
160   *
161   * Runs a set of work item restores.
162   *
163   * Args:
164   *    SubmitObject
165   *        const int CountDrivesAvailable,
168   *        boolean_ly QuitFlag,
169   *        boolean_ly (*CancelRestoreTest)(),
170   *        CancelRestoreTest().
171   *
172   * Returns: int 0 for success
173   */
174  int
175  RunWorkItemRestores(int SubmitObject, boolean_ly (*CancelRestoreTest)(

177  /*
178   * Buckets the work items into trail queues. The trail queues are sorted
179   * in the order which the work items should run.
181   */

182      boolean_ly QuitFlag = FALSE;
183      boolean_ly SubQuit = FALSE;   /* Has the user requested a quit. */
185      boolean_ly SemQuit = FALSE;   /* Have we initiated the quit. */

186      int TrailRestoresRunning = 0; /* The number of trail restores running */

187      int TrailRestoresTotal;       /* The number of trail restores left. */

189      int CountDrivesAvailable = 0; /* The count of drives available. */
191      int CountDrivesInUse = 0;     /* The count of drives in use. */

195      int temp_status;
199      int HighestActiveTrail = 0;   /* The trail queues are ordered from 1 to n. */
```

```c
190  {

192      if(debugmode)
193          (void)the_user_error(0,
194              "DEBUG: Running RunWorkItemRestores for %d trails.",
195              TrailRestoresTotal);

197      /*
198       * Generate the trail queues.
199       * Buckets the work items into trail queues. The trail queues are sorted
200       * in the order which the work items should run.
201       */
202      if(0 != GenerateTrailQueues(SubmitObject,
203                                  &TrailRestoresTotal,
204                                  &temp_status))
205      {
206          (void)the_user_error(0,
207              "Internal error: Cannot generate trail
208              queues, cannot continue.");

209          return -1;
211      }

213      TrailRestoresLeft = TrailRestoresTotal;

215      CountDrivesAvailable = DetermineGlobalDrivesUse(/*SubmitObject*/);

216      /*
217       * This is the start up loop to get the initial work item
218       * restores started.
219       */
220      int submitObjID = 0;
221      int submitedElementID = 0;

222      QuitFlag = CancelRestoreTest();

223      while((CountDrivesInUse < CountDrivesAvailable &&
224             (HighestActiveTrail < TrailRestoresTotal) &&
226             (FALSE == QuitFlag))
228      {
229          /*
230           * Activate the Trail Queue.
231           * This allows the trail queues to be used to
232           * determine the work item restores to run.
233           */
235          HighestActiveTrail++;

237          if(0 != ActivateTrailQueue(HighestActiveTrail,
238                                     &temp_status))
239          {
240              (void)the_user_error(0,
241                  "Internal error: Cannot activate trail
242                  queues[1] for trailid %d, cannot continue.",
243                  HighestActiveTrail);

247              return -1;
248          }
249  }
```

```c
129  2  static int
130  2  DetermineGlobalDriveUse();
132  1
133  1  static int
133  1  test_fd_bug(int fd);
135  1
136  1  static int
137  1  FindFdIllForWkItem(int handle,
138  1                     int *TrailID,
139  1                     int *status);
140  1
141  1  static int
142  1  SendRunningWorkItemsQuit();
143  1
143  1  /* End Stubs */
145  1
145  1  static int
146  1  SelectIt(int nfds,
147  1           fd_set *readfds,
148  1           fd_set *writefds,
149  1           fd_set *exceptfds,
150  1           struct timeval *timeout);
152  1
153  1  static int
154  1  HandleWorkItemRestoreResults(int FromFD,
155  1                               int *TrailID,
156  1                               wi_restore_results *result);
157  1
158  1  static int
159  1  RunWorkItemRestoresForTrail(const int TrailID,
160  1                              const int CountDrivesAvailable,
161  1                              boolean_ty (*CancelRestoreRequest)(),
162  1                              boolean_ty QuitFlag,
163  1                              int *CountDrivesInUse);
164  1
165  1  /*
165  1   *  RunWorkItemRestores()
166  1   *
167  1   *  Runs a set of work item restores.
168  1   *
169  1   *  Args:
170  1   *    Submit2Object
171  1   *    CancelRestoreRequest().
172  1   *
173  1   *  Returns: int 0 for success.
174  1   *
175  1   */
176  1  int
177  1  RunWorkItemRestores(int Submit2Object, boolean_ty (*CancelRestoreRequest)())
178  1  {
178  1      boolean_ty QuitFlag = FALSE;    /* Has the user requested a quit. */
179  1      boolean_ty SentQuit = FALSE;    /* Have we initiated the quit. */
181  1
181  1      int TrailRestoresRunning = 0;   /* The number of trail restores running. */
182  1
182  1      int TrailRestoresLeft;          /* The number of trail restores left. */
183  1
183  1      int TrailRestoresTotal;         /* The number of trail restores total. */
185  1
185  1      int CountDrivesAvailable;       /* The count of drives available. */
186  1      int CountDrivesInUse = 0;       /* The count of drives in use. */
188  1
188  1      int temp_status;
189  1      int HighestActiveTrail = 0;     /* The trail queues are ordered from 1 to n. */
```

```c
190  1      if (debugmode)
192  1          (void)the_user_error(0,
193  1              "DEBUG: Running RunWorkItemRestores.");
194  1
195  1      /*                                            This is the highest trail running */
196  1       *  in the order which the work items should run.
197  1       *                                       The trail queues are sorted
198  1       *  Buckets the work items into trail queues.
199  1       */
200  1
202  2      if(0 != GenerateTrailQueues(SubmitObject,
203  2                                  &TrailRestoresTotal,
204  2                                  &temp_status))
205  2      {
206  2          (void)the_user_error(0,
207  2              "Internal error: Cannot generate trail
                     queues; cannot continue.");
208  2
209  1          return -1;
211  1      }
213  1
213  1      TrailRestoresLeft = TrailRestoresTotal;
215  1
215  1      CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);
216  1
216  1      if(debugmode)
217  1          (void)the_user_error(0,
218  1              "DEBUG: RunWorkItemRestores for %d trails.",
219  2              TrailRestoresTotal);
220  2
221  2      /*
221  2       *  This is the start up loop to get the initial work item
222  2       *  restores started.
223  2       */
224  1
226  2      while((CountDrivesAvailable < CountDrivesTotal) &&
227  2            (HighestActiveTrail < TrailRestoresTotal) &&
228  2            (FALSE == QuitFlag))
228  1      {
230  1          int submitobjID = 0;
231  2          int submitelementID = 0;
231  2
232  2          HighestActiveTrail++;
232  2
233  2          /*
235  2           *  Activate the Trail Queue.
237  2           *  This allows the trail queues to be used to
238  2           *  determine the work item restores to run.
239  2           */
239  2
240  2          if(0 != ActivateTrailQueue(HighestActiveTrail,
242  2                                     &temp_status))
243  2          {
244  2              (void)the_user_error(0,
245  2                  "Internal error: Cannot activate trail
                         queues[1] for trailid %d, cannot continue.",
                         HighestActiveTrail);
246  2
247  2              return -1;
248  2          }
249  2  }
```

```c
    /*
     * This sets the count of running work item and media access concurrency for
     * i.e. The count of running work item restores for this trail.
     *      Today this is one.
     */
    if (0 != SetupDrivesAcquired(HighestActiveTrail, 1, &temp_status))
    {
        (void)the_user_error(0,
            "Internal error: Cannot set drive acquired",
            1) for trailid %d, cannot continue.", HighestActiveTrail);
        return -1;
    }

    /* RunWorkItemRestoresForTrail does its own error logging. */

    if (temp_status == 0)
    {
        return -1;
    }

    /* more work may be be need to recover from this error condition. */

    if (0 > (temp_status = RunWorkItemRestoresForTrail(
                               HighestActiveTrail,
                               CountDrivesRestoreAvailable,
                               CancelRestoreYest,
                               &QuitFlag,
                               &CountDrivesInUse)))

    /* End while() initial startup loop */
    while(1)
    {
        int    HighestFd = 0;
        fd_set WorkItemFromFds;
        int    retStatus;
        struct timeval timeout = {5, 0};

        if (QuitFlag && (!SentQuit))
        {
            SendRunningWorkItemQuit();
            SentQuit = TRUE;
        }
        (void)the_log_state();
        if (!SentQuit)
        {
            "Restore was quit by user. Quitting restore,
             this could take a while.");
        if(0 != getFromSet(&WorkItemFromFds, &HighestFd, &retStatus))
        {
            (void)the_user_error(0,
                "Internal error: Cannot get auxproc result
                 fds, cannot continue.");
```

```c
            return -1;
        }
#endif
#if (debugmode)
        DebugLogFds("The file descriptors to wait on are ",
            &WorkItemFromFds);
#endif
        if (0 > (retStatus = Select(HighestFd + 1,
                                &WorkItemFromFds,
                                NULL, NULL,
                                &timeout)))
        {
            /* error */
            (void)the_user_error(RSRECOVER_MKERR(errno),
                "Internal error: Cannot get auxproc result
                 fds, cannot continue.");
            return -1;
        }
        else if (0 == retStatus)
        {
            QuitFlag = CancelRestoreYest();
        }
        else
        {
#if (debugmode)
            DebugLogFds("Available fds are ",
                &WorkItemFromFds);
#endif
            int ReadyFds = retStatus;
            int FoundFds = 0;
            int index;

            /* If there are available fds then we may want to
             * schedule the next work item restore. We should
             * check if the user initiated a quit.
             */
            QuitFlag = CancelRestoreYest();

            for (index = 0;
                 (index < (HighestFd + 1)) && (FoundFds < ReadyFds);
                 index++)
            {
                int StartWorkItemForTrail = 0;
                if (FD_ISSET(index, &WorkItemFromFds))
                {
                    int TrailID;
                    int TrailAcquired;
                    int wl_restore_results;
                    FoundFds++;

                    memset(&results, 0, sizeof(wl_restore_results));

                    if(0 != HandleWorkItemRestoreResults(index,
                                        &TrailID,
                                        &results))
```

```
371  6          }
372  6              return -1;
373  5          }
374  5
375  5          /* HandleWorkItemRestoreResults will do its own logging */
376  5           * This is where we may want to retry the Work Item
377  5           * Based on if it passes or fails

379  5          CountDrivesInUse--;

382  5          if (0 >
383  5              {StartWorkItemforTrail =
384  5              RunWorkItemRestoresForTrail(TrailID,
385  5                  CountDrivesAvailable,
386  5                  &QuitFlag,
387  5                  CancelRestoreTest,
388  5                  &CountDrivesInUse))
389  6
390  5              /* RunWorkItemRestoresForTrail does its own logging. */
391  5              return -1;
393  5          }
394  5          else if {StartWorkItemforTrail == 0)
395  5          {
396  5              (void)the_user_error(0,
397  5                  "Internal error: Cannot continue. ";
398  5                  number of running work items for trail, cannot determine
399  5                  this trail
400  5                  lets check to see if this is the last work item
401  5                  0 work items started above.
402  6                  Testing for No work items left running or started
403  5                  for this trail.
404  5          int wiCount;

406  6          if(0 != GetRunningWI(TrailID, &wiCount, &temp_status))
407  6          {
408  6              (void)the_user_error(0,
409  7                  "DEBUG: RunWorkItemRestores no
410  8                  more work items left for trailid %d
411  7                  but %d wiCount workitem still running.",
                        TrailID, wiCount);
413  7              if(debugmode)
414  6                  return -1;
415  6          }
416  6
418  6          (void)the_user_error(0,

419  6          if(0 == wiCount) && (0 == StartWorkItemfor
420  7
421  7              TrailRestoresRunning--;
422  7              TrailRestoresLeft--;

424  7              if(0 != DeactivateTrailQueue(TrailID, &temp_status))
425  8              {
426  8                  (void)the_user_error(0,
427  8                      "Internal error: Cannot
                            deactivate trail queue for trailid %d, cannot continue.", TrailID);

429  8                  return -1;
430  8              }
```

```
431  7          /* This test is to determine if
432  7           * there may be another not yet started trail restore finished,
433  7           * drive available the next trail restore will be
434  7           *   started.
435  7
436  7          if (0 != TrailRestoresLeft) &&
437  7              (HighestActiveTrail < TrailRestoresTotal) &&
438  7              (CountDrivesInUse < CountDrivesAvailable))
439  7          {

441  8              (void)the_user_error(0,
443  8                  "Internal error: Cannot set
444  8                  trailid %d, cannot continue.",
445  8                  &temp_status))
446  8
447  9              HighestActiveTrail++;
448  9
449  9              if (0 != ActivateTrailQueue(HighestActiveTrail,
450  9                  1,
451  8                  &temp_status))
453  9              {

455  9                  (void)the_user_error(0,
456  9                      "Internal error: Cannot
457  9                      drive acquired(2) for trailid %d, cannot continue.",
458  9                      &temp_status))
459  9
461  9              if (0 != SetUpDrivesAcquired(
462  9                  HighestActiveTrail, 1, &temp_status))
464  9              {
465  9                  return -1;
466  9              }
467  9
468  9              /* RunWorkItemRestoresForTrail does its own logging */

470  8              if (0 > (temp_status = RunWorkItemRestoresForTrail(
471  7                  HighestActiveTrail,
472  8                  CountDrivesAvailable,
473  7                  &QuitFlag,
474  7                  CancelRestoreTest,
475  7                  &CountDrivesInUse)))
476  8                  return -1;
477  8              }
478  9
479  9              (void)the_log_state(0,
480  8                  "Internal error: Trail %d
481  8                  restore had no work item to run().", HighestActiveTrail);
482  8
483  8              if(temp_status == 0)
                    {
                        return -1;
                    /* If this Trail had no work item we
                       Should attempt to run the next trails
                       work items here. This would be an internal
                       error if a trail queue had no work item
                       restores
                    */
487  9              if(temp_status > 0)
489  9              {
```

```
484      /* If at least one work item was started for this
485       * trail, then we have started a new trail.
486       */
487      TrailRestoresRunning++;
488
489        }
490      }
491    } /* end for() */
492
493  } /* else Available fds */
494
496    /* Terminate the loop if either
498     *
499     * 1] Sent the work items the quit AND
500     *     No Trail restores a running.
501     * OR
502     * 2] No more Trail restores are left.
503     *
504     */
505
507    if((0 == TrailRestoresRunning) && (SentQuit)) ||
508       (0 == TrailRestoresLeft))
509    {
510      break;
511    }
513
     } /* end while(1) */
514  if(0 == "TrailRestoresRunning) && (SentQuit))
515  {
516    (void)rlbe_log_stats(0,
517              "Restore was quit by user.
518                Work item restore quit.");
519  }
520  return 0;
```

```
522      /* Functions needed
523       SendRunningWorkItemsQuit();
524       InterpretWorkItemRestoreResults();
525       */
527  static int
528  Select(int nfds,
529         fd_set *readfds,
530         fd_set *writefds,
531         fd_set *exceptfds,
532         struct timeval *timeout)
533  {
534    int retSelect;
536    do
537    {
538      retSelect = select(nfds,
539                         readfds,
540                         writefds,
541                         exceptfds,
542                         timeout);
544    } while ((-1 == retSelect) && (EINTR == errno));
546    return retSelect;
547  }
```

```
550    openproc
551    InitiateWorkItemRestore (const int SubmitObjID,
552                             const int SubmitEleMID)
553    {
554        struct auxproc AuxprocVitals;
555        errno_ty StartupAPResults = EXIT_FAILURE;
556        time_ty StartTime, EndTime;
557        int TempStatus;
558        char junk_executable[1024];
559        char **junk_argv;
560        char **AP_env = NULL;
561        int SOStatus;
562        char clientName[256] = "";
563        int clientPort;
564        int status;

567        /*
568         * The restore if there are any output variables are ignored.
569         * Let's see if there any environment variables to set.
570         */
571        if(E_SUCCESS != GetSOExecutionPhase(SubmitObjID,
572                                            junk_executable, 1024,
573                                            &junk_argv,
574                                            &AP_env,
575                                            &SOStatus))
576        {
577            (void)lbe_user_error(0,
578                "Internal Error: Could not get environment
                     variables.");

580            return -1;
581        }

583        if (E_SUCCESS = GetSERemdConnect(SubmitObjID,
584                                         SubmitEleMID,
585                                         clientName, 256,
586                                         &clientPort))
587        {
588            (void)lbe_user_error(0,
589                "Internal Error: Could not get Remote Client name
                     .");

591            return -1;
592        }

595        StartupAPResults = StartupAuxprocess(0 /* XXX */,
                                                &AuxprocVitals,
                                                AP_env,
                                                clientName,
                                                clientPort);

598        if (StartupAPResults != StartupAPResults)
599        {
600            (void)lbe_user_error(0,
                    "port to connect.");

602            return -1;
603        }
604        else
605        {
606            /* StartupAuxprocess does its own logging. */
607            return i;
608        }

610        /*
611         * We need to close the bulk fd. This file descriptor
612         * is not being used any more. If we do not close it
613         * here we will have a file descriptor leak because
614         * we won't be able to determine what it was when the
615         * work item completes.
616         */
```

```
612        close(AuxprocVitals.xp_fd_bulk_to_x);

614        time(&StartTime);

616        if(0 != newHandleSet(AuxprocVitals.xp_fd_to_x,
                                AuxprocVitals.xp_fd_item_x,
                                AuxprocVitals.xp_fd_prog_from_x,
                                SubmitObjID,
                                SubmitEleMID))
           {
                (void)lbe_user_error(
                    0, "Internal Error: Could not register handle set.");

627             return -1;
628         }

629        if(0 > StartWorkItemRestore(&rcp,
                                       AuxprocVitals,
                                       SubmitObjID,
                                       SubmitEleMID))
           {
                (void)lbe_user_error(
                    0, "Error in StartWorkItemRestore  SubmitObjID %d."
                    "SubmitEleMID %d",SubmitEleMID);

641        /*
642         * the following code kills auxproc when recx or xpio do not
                   start
643         * if errors occur in the detention or KillWorkItemRestore the
                   messages are
644         * logged in those calls, plus,
646         * we already know there was an error and that
647         * is why we are doing this right now.
648         */
649        time(&EndTime);

651        deleteHandleSet();

653        KillWorkItemRestore(
                AuxprocVitals.xp_fd_from_x, EndTime, EP_RB_RECOVER_ALLFAIL,
                AuxprocVitals.xp_pid, AuxprocVitals.xp_fd_to_x);

655            return -1;

658        }

660    }  /* InitiateWorkItemRestore */
```

```c
663        *       interprate return.
664        *       Drain progress.
665        *       Send final Progress for work item.
666        *       Delete the handle set.
667        */
668
     static int
     HandleWorkItemRestoreResults(int FromFD,
                                  int *TrailID,
                                  wi_restore_results *results)
669 1   {
670 1       int ret = 0;
671 1       int retries = 0;
672 1       int GetAuxprocResults;
673 1       int GetAuxprocResultsStatus;
674 1       int DrainInResult;
675 1       int DrainedFD;
676 1       int wiCount;
677 1       int AuxProcPid;
678 1       int ToFD, getFromFD, ProgressFD;
679 1       time_t timeout, jobstat;
680 1       unsigned long timeout;
681 1       int timeout = 3; /* Lets try 3 seconds */
682 1       boolean_ty fromDHangup = FALSE;
683 1
684 1       ToFD = getFromFD = ProgressFD = -1;
685 1
686 1       while(! (fromDHangup))
687 1       {
688 2           GetAuxprocResultsStatus = GetAuxprocResults(FromFD, results);
689 2
690 2           if(-1 == GetAuxprocResultsStatus)
691 2           {
692 3               /* GetAuxprocResults() does its own logging */
693 3
694 3               (void)the_user_error(0, "Error in GetAuxprocResults");
695 3               return -1;
696 3           }
697 2
698 2           if(0 == GetAuxprocResultsStatus)
699 3           {
700 3               if(test_fd_hup(FromFD) == 1)
701 3               {
702 4                   fromDHangup = TRUE;
703 4               }
704 3
705 3               /* The remote result are not always going to
706 4                * be set. For example if the remote command
707 4                * is not started correctly.
708 3                */
709 3
710 3               if(results->local_exit_set == TRUE)
711 2               {
712 2                   break;
713 2               }
714 3               else
715 2               {
716 2                   sleep(1);
717 3                   test_fd(FromFD);
718 2                   continue;
719 3               }
720 3           }
```

```c
727 1       time(&EndTime);
728 1
729 2       if(0 != PushDrainRequest(FromFD, &TempStatus))
730 2       {
731 2           (void)the_user_error(0,
732 2               "Internal error: Could not push drain"
733 1               " request, cannot continue.");
734 1           return -1;
735 2       }
736 1
737 2       if(0 != FindTrailQueueOfWI(FromFD, TrailID, &TempStatus))
738 2       {
739 2           (void)the_user_error(0,
740 1               "Internal error: Could not find trail id for"
741 2               " finished work item, cannot continue.");
742 2           return -1;
743 1       }
744 1
745 2           return -1;
746 1       }
747

748    /* Lets give the progress thread a chance to drain keeping busy in
749       the meanwhile.
750    */
751 2       if(0 != DecrementRunningWI(*TrailID, &wiCount, &TempStatus))
752 2       {
753 1           return -1;
756 2           (void)the_user_error(0,
757 2               "Internal error: Could not decrement running"
758 2               " work items for trail, cannot continue.");
759 2           return -1;
761 1       }
762 1
764 2       if(0 != gethandleSet(
765 2               &ToFD, &getFromFD, &ProgressFD, &TempStatus))
766 2       {
767 2           (void)the_user_error(0,
768 1               "Internal error: Could not get auxproc file"
769 2               " descriptors for work item, cannot continue.");
770 2           return -1;
771 1       }
772 2       if(FromFD != getFromFD)
773 2       {
774 2           (void)the_user_error(0,
775 2               "Internal error: mismatch on from file"
776 1               " descriptors for work item, cannot continue.");
778 1           return -1;
779 1
780 1       while (0 != (ret = PopDrainResult(timeout,
781 1                   &DrainResult,
782 1                   &DrainedFD,
783 1                   &DrainedP,
784 1                   &TempStatus)) && retries < 3)
            {
                if (ret != 0)
```

```
786 2        (void)the_user_error(0,
787 2                  "Internal error: Could not pop drain results,
788 1                          cannot continue.");
789 1
790 1        return -1;
790 1    }
794 2
794 1    /*
795 2     *  Translate the local and remote error statuses
796 2     *  to an sperrno value:
797 1     */
798 1
800 2    if(0 != results->local_exit_status)      /* use local error, if any */
801 2        switch (results->local_exit_status)
802 2        {
804 2        case XQ_EXIT_ALLFAIL:
805 2            jobstat = EP_RB_RECOVER_ALLFAIL;
806 2            break;
807 2        case XQ_EXIT_MANYFAIL:
808 2            jobstat = EP_RB_RECOVER_MANYFAIL;
809 2            break;
810 2        case XQ_EXIT_FEWFAIL:
811 2            jobstat = EP_RB_RECOVER_FEWFAIL;
812 2            break;
813 2        case SPEXIT_REMOTE_STDERR_PROTOCOL:
814 2        case SPEXIT_REMOTE_STDERR_FAIL:
815 2            jobstat = EP_RB_RECOVER_CLIENTFAIL;
816 2            break;
817 2        case XQ_EXIT_STOPPED:           /* treat like signal */
818 2            if ( XQ_EXIT_SIGBASE < results->local_exit_status
819 2               || XQ_EXIT_STOPPED == results->local_exit_status
820 2               /* killed by signal or stopped: separate error for sigpipe */
821 1               if( XQ_EXIT_SIGBASE + SIGPIPE == results->local_exit_status )
822 1                   jobstat = EP_RB_RECOVER_SERVER_SIGPIPE;
823 2               else
824 2                   jobstat = EP_RB_RECOVER_SERVER_SIGNAL;
825 2
826 2            else
827 2            {
828 2                /* generic server failure, unless client failed too */
829 2                jobstat = results->local_exit_status
830 1                    if (0 != results->remote_exit_status )
831 1                        jobstat = EP_RB_RECOVER_CLIENTFAIL;
832 2            }
832 2        }
832 1        else if( (0 != results->remote_exit_status) ||
834 1                 (0 != results->local_exit_status ) )
835 2        else
836 2            jobstat = E_SUCCESS;
841 2    /* check for signal termination vs all generic
842 2              failures */
843 2
843 2    int    status=0;
844 2    int    rc=0;
845 2    char   *winame=NULL;
846 2    char   *templateName=NULL;
847 2    char   *trailsetName=NULL;
849 2    rc = getHandleSetInformation(FromFD,
                                    &templateName,
                                    &winame,
```

```
850 2                                    &trailsetName,
851 2                                    &status)
852 2        the_log_status(0, "Restore Failure",
853 2                "top level object: %s, template %s.",
854 2                STR_SURE(winame),
855 1                STR_SURE(templateName) );
856 2        free(templateName);
857 2        free(winame);
858 2        free(trailsetName);
859 1    }
860 1
861 2    if (0 == deletehandleclose(FromFD, EndTime, jobstat, &TempStatus))
863 2        (void)the_user_error(0,
864 2                "Internal error: Could not delete Handle
865 1                        Set, cannot continue.");
866 2
867 1        return -1;
868 1    }
869 2    if(0 != killWorkItemRestore(AuxProcPid,
870 2                    -1  /* Hack this arg is not needed yet
871 1                            cmd_to */)
872 2        (void)the_user_error(0,
873 2                "Internal error: Could not kill finished
874 1                        auxproc, cannot continue.");
875 1
876 2        return -1;
877 1    }
878 1    close(ToFD);
879 1    close(FromFD);
879 1    close(StderrFD);
881 2    if(debugmode)
882 2        (void)the_user_error(0,
883 2                "DEBUG: HandleWorkItemRestoreResults Auxproc
884 1                PID %d) just finished for trailid %d work items left = %d.",
885 2                AuxProcPid,
886 2                *trailid,
887 2                wiCount);
889 2        (void)the_user_error(0,
890 2                "DEBUG: HandleWorkItemRestoreResults Auxproc
891 1                PID %d) results are local: %d set's remote: %d set:%s.",
892 2                results -> local_exit_status,
893 2                results -> remote_exit_status,
894 2                results -> local_exit_status ? "TRUE" : "FALSE",
895 2                results -> remote_exit_set ? "TRUE" : "FALSE");
898 1
899 1        return 0;
900 1    }  /* End HandleWorkItemRestoreResults() */
```

```c
904   *
906   *  RunWorkItemRestoresForTrail()
907   *
908   *  Description
909   *      This function starts all the work item for the
910   *      trail. For no this is set to one but concurrency
911   *      will can be supported.
912   *
913   *  Args:
914   *      (I)  TrailID  -- The id for this trail
915   *      (I)  CountDrivesAvailable -- the total drives available to restore
916   *      (I)  QuitFlag -- indicate whether the user has quit the restore.
917   *      (0)  CountDrivesStarted -- The count of trails in use by restore.
921   *      returned.
922   *
923   *  Return int
924   *      if -1 then an error has occurred.
925   *      if 0 or greater then the number of trail restores started will be
926   *
927   */
928  static int
929  RunWorkItemRestoresForTrail(const int TrailID,
930                              const int CountDrivesAvailable,
931                              boolean_t *CancelRestoreCrest(),
932                              boolean_t *QuitFlag,
933                              int *CountDrivesInUse)
934  {
935      int DriveAcquiredForTrail;
936      int DriveConcurrencyForTrail;
937      int submiteObjID;
938      int submiteElementID;
939      int popResults = 0;
940      int temp_status;
941      int CountOfWorkItemRestoresStarted = 0;
941      int wiCount;

941      while(1)
941      {
944          (*CountDrivesInUse)++;

919          if(0 != (popResults = PopWIFromTrailQueue(TrailID,
                                                       &submiteObjID,
                                                       &submiteElementID,
                                                       &temp_status)) &&
                 (SCHED_NO_MORE_JOBS == temp_status))
938          {
937              (void)the_user_error(0,
935                  "Internal error: Cannot pop work item off
                      trail queue, cannot continue.");
933              return -1;
932          }

930          if((-1 == popResults) && (SCHED_NO_MORE_JOBS == temp_status))
                 return;

962          temp_status = InitiateWorkItemRestore(
                              submiteObjID, submiteElementID);

964          if(temp_status != 0)
```

```c
965          {
966              /* InitiateWorkItemRestore() does its own logging */
967              (void)rbe_user_error(0, "Error in InitiateWorkItemRestore,
968                  submiteObjID  %d, submiteElementID %d",
                     submiteObjID, submiteElementID);

970              return -1;
972          }

973          if(0 != IncrementRunningWiT(TrailID, &wiCount, &temp_status))
974          {
975              (void)rbe_user_error(0,
                     "Internal error: cannot increment
                      running work items for trail, cannot continue.");
976              return -1;
977          }

979          CountOfWorkItemRestoresStarted++;

982          if(0 != GetRQDriveAcquired(TrailID,
                     &DriveAcquiredForTrail,
                     &temp_status))
983          {
984              (void)the_user_error(0,
                     "Internal error: Cannot get drives
                      acquired, cannot continue.");
985              return -1;
988          }

991          if(0 != GetRQDriveConcurrency(TrailID,
                     &DriveConcurrencyForTrail,
                     &temp_status))
992          {
993              (void)the_user_error(0,
                     "Internal error: Cannot get drive
                      concurrency, cannot continue.");
994              return -1;
996          }

997          *QuitFlag = CancelRestoreCrest();

1000         if(DriveAcquiredForTrail < DriveConcurrencyForTrail &&
1002             (*CountDrivesInUse < CountDrivesAvailable) &&
1003             (FALSE == *QuitFlag))
1004         {
1005             continue;
1006         }
1007         else
1008         {
1009             break;
1010         }

1013     return CountOfWorkItemRestoresStarted;
1014  }  /* RunWorkItemRestoresForTrail() */
```

```
1018    /* Stub */
1019    static int DetermineGlobalDriveUse()
1020    {
1021    /* Limiting to MAXINT === not limiting... Need resource management
1022       to do this property.
           NOTE: This should now work like eb_dc_restore does.
        */
1023        return MAXINT;
1024
1025    }
```

```
1028    static int
1029    SendRunningWorkItemsQuit()
1030    {
1031        int *APlist;
1032        int count;
1033        int status;
1034        int index;

1036        if(0 != getPIDlist(&count, &APlist, &status))
1037        {
1038            (void)the_user_error(0,
                    "Internal error: Cannot get auxproc pid list,
                     cannot continue.");
1039
1040            return -1;
1041        }

1043        for(index = 0; index < count; index++)
1044        {
1045            QuitWorkItemRestore(APlist[index]);
1046        }
1047        return 0;
1048    }
```

```
1050      /*
1051       * Stub this out for now.
1052       */
1053      static int
1054      InterpretWorkItemRestoreResults(wi_restore_results *results)
1055    1 {
1056    1     return 0;
1057    1 }
```

```
1059      static void
1060      DebugLogFds(char *error_msg,
1061                  fd_set *fds)
1062    1 {
1063    1     int index, fd_count = 0;
1064    1     char buffer[4096];
1065    1     char *bufptr = (char*)buffer;

1068    1     for(index=0;
1069    1         index < 1024;
1070    1         index++)
1071    2     {
1072    2         if( FD_ISSET(index, fds))
1073    3         {
1074    3             int size = 0;
1075    3             size = sprintf(bufptr, "%d,", index);
1076    3             bufptr += size;
1077    3             fd_count++;
1078    2         }
1079    1     }
1080    1     rbc_log_stats(0, "%s fd_count:: %d :: {%s}\n",
1081    1                   error_msg, fd_count, buffer);
1082    1 }
```

```
1085
1086
1087 1   static int
1088 1   test_fd(int fd)
1089 1   {
1090 1       fd_set read_fd;
1091 1       int ret_select;
1092 1       struct timeval timeout = {0, 0};
1093
1094 1       FD_ZERO(&read_fd);
1095 1       FD_SET(fd, &read_fd);
1096 1
1097 2       do
1098 2       {
1099 2           ret_select = select(fd + 1, &read_fd, NULL, NULL, &timeout);
1100 1       } while((-1 == ret_select) && (EINTR == errno));
1101
1102 1       return ret_select;
1103
1104     }
```

```
1106     /*
1107      * test_fd_hup()
1108      *
1109      * Description: Test the supplied file descriptor to check for the hang up
1110      *   if it has had the hang up condition.
1111      *
1112      *   Args:  Input  fd -- the file descriptor to check for the hang up condition.
1113      *
1114      *   Returns:
1115      *     1 for HUP event received on fd.
1116      *     0 No HUP event received on fd.
1117      *    -1 errno set.
1118      */
1119     static int
1120     test_fd_hup(int fd)
1121 1   {
1122 1       struct pollfd fds;
1123 1       int ret_poll;
1124
1125 2       if(fd < 0)
1126 2       {
1127 2           errno = EINVAL;
1128 2           return -1;
1129 1       }
1130
1131 1       fds.fd = fd;
1132 1       fds.events = POLLHUP;
1133 1       fds.revents = 0;    /* initialize */
1134
1135 2       do
1136 2       {
1137 2           ret_poll = poll(&fds, 1, 0);
1138 1       } while((-1 == ret_poll) && (EINVAL == errno));
1139
1140 2       if(-1 == ret_poll)
1141 1           return -1;
1142
1143 2       if(POLLHUP & fds.revents)
1144 1           return 1;
1145 1       else
1146 1           return 0;
1147
1148     } /* end test_fd_hup() */
```

RSLwisvr.c 25

RSLwisvr.c 26

```
  1   /*
  2   **
  3   **      Copyright 1996, 1997 EMC Corporation
  4   **
  5   */
  6
  7   /*
  8   **      EDMRESchedApi.cc
  9   **
 10   **      Mission Statement:  file that contains an API to manage the order
 11   **                          in which
 12   **
 13   **      Primary Data Acted On:
 14   **                          restores occur
 15   **
 16   **      Compile-Time Options:
 17   **
 18   **      Basic idea here:    A few calls to manage the order in which work
 19   **                          items
 20   **                          are run.
 21   **                          Currently things are ordered by where they
 22   **                          appear in the Submit list but this will need to
 23   **                          change in the future.
 24   **
 25   */
 26
 27   #endif
 18   #if !defined(lint)
      static char    RCS_id [] = "@(#)$RCSfile: EDMRESchedApi.cc,v $ *
                                 "$Revision: 1.0 $ *
                                 "$State: 1997/02/06 20:49:15 $";
 17   #endif
 18
 19   #include <sel/c_portable.h>
 20   #include <sel/cp_xopen.h>
 21   #include <sel/i_inout.h>
 23
 24   #include <stdlib.h>
 26   #include <sys/types.h>
 27   #include <pthread.h>
 29
 30   // Rogue Wave includes
 31   #include <rw/collect.h>
 33   #include <rw/rwfile.h>
 34   #include <rw/vstream.h>
 35   #include <rw/rwfile.h>
 37   #include <rw/bintree.h>
 38
 39   #include <restore/REDrogemsg.h>
 40   #include <restore/dispatch_daemon.h>
 41
 43   #include <restore/EDMRESubmitApi.h>
 44   #include <EDMREHandleMgrApi.h>
 46   #include <EDMRESchedulecWT.h>
 47   #include <EDMRETaskFile.h>
 48   #include <EDMRESchedApi.h>
 49
 50   typedef struct {
 52       char        templatename[TEMPLNAME_SIZE];
 53       boolean_ty  alternate;
 54       int         trailnum;
 56   } findArg;
 57
 58   static unsigned int    numberOfQueues = 0;
 59   static RWBinaryTree    trailllists;
 60   static pthread_mutex_t  G_scheduleMtx = PTHREAD_MUTEX_INITIALIZER;
 62
```

```
 63   /*
 64   **      Routine:    LockScheduleMutex
 65   **
 66   **      Inputs:     None
 67   **
 68   **      Outputs:    None
 69   **
 70   **      Return Codes:
 71   **                  None
 72   **
 73   **      Purpose:    Lock the mutex for the handle tree object
 74   **
 75   */
 76
 78   static void
 79   LockScheduleMutex()
 80   {
 81       static boolean_ty first = TRUE;
 82
 83       if (first == TRUE)
 84       {
 85           first = FALSE;
 86           pthread_mutex_init(&G_scheduleMtx, NULL);
 87       }
 88
 89       pthread_mutex_lock(&G_scheduleMtx);
 90   }
```

```
 92
 93   **
 94   ** Routine:    UnlockScheduleMutex
 95   **
 96   ** Inputs:     None
 97   **
 98   ** Outputs:    None
 99   **
100   ** Return Codes:
101   **      None
102   **
103   ** Purpose:   Unlock the mutex for the handle tree object
104   **
105   *********************************************************************
106   */
108   static void
109   UnlockScheduleMutex()
110 1 {
111 1   pthread_mutex_unlock(&G_schedulemtx);
112 1 }
```

```
114   /*******************************************************************
115   **
116   ** Routine:   LookupSchedWI
117   **
118   ** Inputs:    int ID - trail object ID associated with element
119   **            int jobID - WI job ID associated with element
120   **
121   ** Outputs:   int *status - status of the function if an error occured
122   **            EDMRESchedWI **wi - place to put the pointer to the
123   **                               element
124   **
125   ** Return Codes:
126   **       int - 0 for success or non-zero for failure
127   **
128   ** Purpose:   Finds a scheduled work item based on the trail ID and the
129   **            work item ID.
130   **
131   *******************************************************************
132   */
133   static int
134   LookupSchedWI(int ID, int jobID, EDMRESchedWI **wi,
135                 int *status)
136 1 {
137 1   EDMRETraillist *trl;
138 1   EDMRETraillist *ret;
139
140 1   if (status == NULL)
141 2   {
142 2     return -1;
143 1   }
144
145 1   if (wi == NULL)
146 2   {
147 2     *status = SCHED_BAD_PARAM;
148 2     return -1;
149 1   }
150
151 1   trl = new EDMRETraillist();
152
153 1   if (trl == NULL)
154 2   {
155 2     *status = SCHED_NO_MEMORY;
156 2     return -1;
157 1   }
158
159 1   trl -> setTrailID(ID);
160
161 1   LockScheduleMutex();
162
163 1   ret = (EDMRETraillist *) traillists.find(trl);
164
165 1   delete trl;
166
167 1   if (ret == NULL)
168 2   {
169 2     *status = SCHED_TRAIL_LOOKUP_FAILED;
170 2     UnlockScheduleMutex();
171 2     return -1;
172 1   }
173
174 1   *wi = ret -> getSchedWI(jobID);
```

```
176 1    UnlockScheduleMutex();

178 1    if (*wi == NULL)
179 2    {
181 3        *status = SCHED_JOB_LOOKUP_FAILED;
182 4        return -1;
         }

184 1    return 0;
185 }
```

```
187  /*********************************************************
188  **
189  ** Routine:    LookupTrailObject
190  **
191  ** Inputs:     int ID - trail object ID
192  **
193  ** Outputs:    int *status - status of the function if an error occured
194  **
195  ** Return Codes:
196  **             int  - 0 for success for failure
197  **             int  - -1 for success for failure
198  **
199  ** Purpose: Finds a trail object based on the trail ID.
200  **
             *********************************************************
203  */
204  static int
205  LookupTrailObject(int ID, EDMRETrailList **trl, int *status)
206  {
207  1    EDMRETrailList *tmptrl;
208  1    EDMRETrailList *ret;

209  1    if (status == NULL)
210  2    {
211  2        return -1;
212  1    }

214  1    if (trl == NULL)
215  2    {
216  2        *status = SCHED_BAD_PARAM;
217  2        return -1;
218  1    }

220  1    tmptrl = new EDMRETrailList();

222  1    if (tmptrl == NULL)
223  2    {
224  2        *status = SCHED_NO_MEMORY;
225  2        return -1;
226  1    }

228  1    tmptrl -> setTrailID(ID);

230  1    LockScheduleMutex();

232  1    ret = (EDMRETrailList *) traillists.find(tmptrl);

234  1    UnlockScheduleMutex();

236  1    delete tmptrl;

238  1    if (ret == NULL)
239  2    {
240  2        *status = SCHED_TRAIL_LOOKUP_FAILED;
241  2        return -1;
242  1    }

244  1    *trl = ret;

246  1    return 0;
247  }
```

```c
249  /************************************************
250  **
251  ** Routine:    NewTrailObject
252  **
253  ** Inputs:     NONE
254  **
255  ** Outputs:    int *status - status of the function if an error occured
256  **
257  ** Return Codes:   int - ID of the new trail object
258  **
259  **
260  **
261  ** Purpose:    Creates a new trail object and inserts it in the trail
262  **             list.
263  **
264  ************************************************/
265
266  int
267  NewTrailObject(int *status)
268  {
269      EDMRETraillist *tl;
270      EDMRETraillist *ret;
271
272      if (status == NULL)
273      {
274          return 0;
        }
276
        tl = new EDMRETraillist();
278
279      if (tl == NULL)
280      {
281          *status = SCHED_BAD_PARAM;
282          return 0;
        }
284
        tl -> setTrailQID(++numberOfQueues);
286
        LockScheduleMutex();
288
        ret = (EDMRETraillist *) traillists.insert(tl);
290
        UnlockScheduleMutex();
292
294      if (ret == NULL)
295      {
296          *status = SCHED_TRAIL_INSERT_FAILED;
297          delete tl;
            return 0;
        }
299
300      return numberOfQueues;
    }
```

```c
302  /************************************************
303  **
304  ** Routine:    NewSchedWI
305  **
306  ** Inputs:     int ID - trail ID associated with new element
307  **             int submitID - submit ID associated with new element
308  **             int elementID - submit element ID associated with new
309  **                             element
310  **
311  ** Outputs:    int *status - status of the function if an error occured
312  **
313  ** Return Codes:   int - ID of the new sched WI tree.
314  **
315  **
316  ** Purpose:    Creates a new scheduled work item element and inserts it
317  **             in the
318  **
319  ************************************************/

321  int
322  NewSchedWI(int ID, int submitID, int elementID, int *status)
323  {
324      EDMRETraillist *tl;
325      EDMRETraillist *ret;
326      EDMREScheduledWI *wi;
327      int winum = 0;

329      if (status == NULL)
330      {
331          return 0;
332      }

334      tl = new EDMRETraillist();

336      if (tl == NULL)
337      {
338          *status = SCHED_BAD_PARAM;
339          return 0;
340      }

342      tl -> setTrailQID(ID);

344      LockScheduleMutex();

346      ret = (EDMRETraillist *) traillists.find(tl);

348      delete tl;

350      if (ret == NULL)
351      {
352          *status = SCHED_TRAIL_LOOKUP_FAILED;
353          UnlockScheduleMutex();
354          return 0;
355      }

357      winum = ret -> newScheduledWI();

359      if (winum <= 0)
360      {
361          *status = SCHED_NEW_JOB;
362          UnlockScheduleMutex();
```

```
363  2          )
364  1        }

366  1        wi = ret -> getSchedWI(wiNum);

368  1        UnlockScheduleMutex();

370  1        if (wi == NULL)
371  2        {
372  2          *status = SCHED_JOB_LOOKUP_FAILED;
373  2          return 0;
374  1        }

376  1        wi -> setSubmitObjsectID(submitID);
377  1        wi -> setSubmitElementID(elementID);
379  1
380  1        return wiNum;
             }
```

```
382  /***************************************************
383   **
384   **  Routine:    findTrail
385   **
386   **  Inputs:     findArg - Traillist to check against
387   **
388   **  Outputs:    findArg - structure containing SubmitElement and status
389   **
390   **
391   **  Return Codes:
392   **              None
393   **
394   **  Purpose:    Sets the status to non-zero if a match is found.
395   **
396   ****************************************************/

398  static void
399  findTrail(IN RWCollectable* c, IN void *f)
400  {
401      EDMRESTraillist *trl = (EDMRESTraillist *) c;
402      findArg *fa = (findArg *) f;
403      char trItemp[TEMPLNAME_SIZE];

405      trl -> getTemplateName(trItemp, sizeof(trItemp));

407      if (strcmp(fa->templatename, trItemp) == 0 &&
408          fa -> alternate == trl -> isAlternateTrailset())
409      {
410          fa -> trainum = trl -> getTrailID();
411      }
412  }
```

```c
/*
** ***********************************************************
**
** Routine:    GenerateTrailQueues
**
** Inputs:     int ID - submit ID
**
** Outputs:    int *status - status of the function if an error occurred
**             int *trailcount - place to put the trail queue count
**
** Return Codes:
**             int - 0 if success and non-zero otherwise
**
** Purpose:    Generate the schedule ordered by trail.
**
** ***********************************************************
*/
int
GenerateTrailQueues(int ID, int *trailcount, int *status)
{
    EDMRESubmitObj *so;
    EDMRESubmitElement *se;
    EDMRETrailList *trl;
    int ret;
    int i;
    int numElements;
    findAny  fa;

    if (status == NULL)
    {
        return -1;
    }

    if (trailcount == NULL)
    {
        return -1;
    }

    if (trailcount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupSubmitObject(ID, &so, status);

    if (ret != 0)
    {
        return ret;
    }

    numElements = so -> getWtCount();

    for (i = 1; i < numElements + 1; i++)
    {
        se = so -> getSubmitElement(i);

        if (se == NULL)
        {
            *status = SCHED_SE_LOOKUP_FAILURE;
            break;
        }

        fa.alternate = se -> IsAlternateTrailset();
        se -> getTemplate(fa.templatename, TEMPLATE_NAME_SIZE);
        fa.trailnum = 0;
```

```c
        trailList.apply(findTrail, &fa);

        UnlockScheduleMutex();

        if (fa.trailnum == 0)
        {
            char  setImpl[TEMPLATE_NAME_SIZE];

            fa.trailnum = NewTrailObject(status);

            if (fa.trailnum == 0)
            {
                break;
            }

            ret = LookupTrailObject(fa.trailnum, &trl, status);

            if (ret != 0)
            {
                break;
            }

            trl = setSubmitID(ID);
            trl -> setAlternateToTrailset(se -> IsAlternateTrailset());

            se -> getTemplate(setImpl, sizeof(setImpl));
            trl -> setTemplateName(setImpl);

            ret = NewSchedWL[fa.trailnum, ID, i, status);

            if (ret <= 0)
            {
                break;
            }
        }
    }

    *trailcount = numberOfQueues;

    return 0;
}
```

```
520     /*********************************************
521     **
522     ** Routine:    PurgeTrailQueue
523     **
524     ** Inputs:     None
525     **
526     ** Outputs:    None
527     **
528     ** Return Codes:
529     **             None
530     **
531     ** Purpose:    Purge all lists.
532     **
533     **
534     */
        void
        PurgeTrailQueue()
536     {
537        LockScheduleMutex();
538 1
539 1      trailLists.ClearAndDestroy();
541 1
543 1      UnlockScheduleMutex();
545 1
546 1      numberOfQueues = 0;
        }
```

```
548     /*********************************************
549     **
550     ** Routine:    ActivateTrailQueue
551     **
552     ** Inputs:     int ID    - trail ID
553     **             int drivecount - number of drives to use
554     **
555     ** Outputs:    int *status - status of the function if an error occured
556     **
557     ** Return Codes:
558     **             int - 0 if success and non-zero otherwise
559     **
560     ** Purpose:    Active the given trail for restore.
561     **
562     **
563     */
        int
        ActivateTrailQueue(int ID, int drivecount, int *status)
565     {
566        int    ret = 0;
568
569        EDMRESTrailList *trl;
571 1
571 1      if (status == NULL)
572 2      {
572          return -1;
573 1      }
574
576 1      ret = LookupTrailObject(ID, &trl, status);
578
578 1      if (ret != 0)
579 2      {
580          return ret;
580 1      }
581
583 1      if (trl == NULL)
584 2      {
584          return -1;
585 1      }
586
588 1      trl -> setTrailActive(TRUE);
589 1      trl -> setMaxDrives(drivecount);
591
591 1      return 0;
592     }
```

```
594   /*
595    **************************************************************
596    **
597    ** Routine:    DeactivateTrailQueue
598    **
599    ** Inputs:     int ID - trail ID
600    **
601    ** Outputs:    int *status - status of the function if an error occured
602    **
603    ** Return Codes:
604    **             int - 0 if success and non-zero otherwise
605    **
606    ** Purpose:    Deactivate the given trail for restore.
607    **
608    **************************************************************
       */
610   int
611   DeactivateTrailQueue(int ID, int *status)
612   {
613       EDMRETrailList *trl;
614       int ret = 0;

616       if (status == NULL)
617       {
618           return -1;
619       }

621       ret = LookupTrailObject(ID, &trl, status);

623       if (ret != 0)
624       {
625           return ret;
626       }

628       if (trl == NULL)
629       {
630           return -1;
631       }

633       trl -> setTrailActive(FALSE);

635       return 0;
636   }
```

```
638   /*
639    **************************************************************
640    **
641    ** Routine:    SetTODriveConcurrency
642    **
643    ** Inputs:     int ID - trail ID
644    **             int drivecount - number of drives to use
645    **
646    ** Outputs:    int *status - status of the function if an error occured
647    **
648    ** Return Codes:
649    **             int - 0 if success and non-zero otherwise
650    **
651    ** Purpose:    Set the drive concurrency for the given trail.
652    **
653    **************************************************************
       */
655   int
656   SetTODriveConcurrency(int ID, int drivecount, int *status)
657   {
658       EDMRETrailList *trl;
659       int ret = 0;

661       if (status == NULL)
662       {
663           return -1;
664       }

666       ret = LookupTrailObject(ID, &trl, status);

668       if (ret != 0)
669       {
670           return ret;
671       }

673       if (trl == NULL)
674       {
675           return -1;
676       }

678       trl -> setMaxDrives(drivecount);

680       return 0;
681   }
```

```
683
684     /* ***********************************************************
685     **
686     ** Routine:   GetTODriveConcurrency
687     **
688     ** Inputs:    int ID - trail ID
689     **
690     ** Outputs:   int *status - status of the function if an error occured
691     **            int *drivecount - place for the number of drives to use
692     **
693     ** Return Codes:
694     **            int - 0 if success and non-zero otherwise
695     **
696     ** Purpose:   Retrieve the number of drives to use for the given trail
697     **            for restore.
698     **
699     ********/
701     int
702     GetTODriveConcurrency(int ID, int *drivecount, int *status)
703     {
704 1       int  ret = 0;
705 1
704 1       EDMRETrailList *trl;
705 1
707 1       if (status == NULL)
708 2       {
709 2           return 1;
710 1       }
712 1       if (drivecount == NULL)
713 2       {
714 2           return 1;
715 1       }
716 2
718 1       ret = LookupTrailObject(ID, &trl, status);
720 1       if (ret != 0)
721 2       {
722 2           return ret;
723 1       }
725 1       if (trl == NULL)
726 2       {
727 2           *status = SCHED_BAD_PARAM;
728 1           return -1;
728 1       }
730 1       *drivecount = trl -> getMaxDrives();
732 1       return 0;
733     }
```

```
735
736     /* ***********************************************************
737     **
738     ** Routine:   SetTODrivesAcquired
739     **
740     ** Inputs:    int ID - trail ID
741     **            int drivecount - the number of drives in use
742     **
743     ** Outputs:   int *status - status of the function if an error occured
744     **
745     ** Return Codes:
746     **            int - 0 if success and non-zero otherwise
747     **
748     ** Purpose:   Sets the number of drives in use for the given trail
749     **            for restore.
750     **
751     ******/
753     int
754     SetTODrivesAcquired(int ID, int drivecount, int *status)
755     {
756 1       int  ret = 0;
757 1
759 1       EDMRETrailList *trl;
760 2
761 2       if (status == NULL)
762 1       {
764 1           return -1;
766 1       }
767 2
768 2       ret = LookupTrailObject(ID, &trl, status);
769 1
771 1       if (ret != 0)
772 2       {
773 2           return ret;
774 1       }
776 1       if (trl == NULL)
778 2       {
779 2           return -1;
        }
        trl -> setDrivesInUse(drivecount);
        return 0;
    }
```

```
/*
**********************************************************
**
** Routine:     GetTODrivesAcquired
**
** Inputs:      int ID - trail ID
**
** Outputs:     int *status - status of the function if an error occured
**              int *drivecount - a place to put the number of drives in
**                                use
**
** Return Codes:
**              int - 0 if success and non-zero otherwise
**
** Purpose:     Gets the number of drives in use for the given trail
**              for restore.
**
**********************************************************
*/
int
GetTODrivesAcquired(int ID, int *drivecount, int *status)
{
    EDMRESETrailList *trl;
    int ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    if (trl == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    *drivecount = trl -> getDrivesInUse();

    return 0;
}
```

```
/*
**********************************************************
**
** Routine:     IncrementRunningWI
**
** Inputs:      int ID - trail ID
**
** Outputs:     int *status - status of the function if an error occured
**              int *wiCount - number of work items running after
**                             increment.
**
** Return Codes:
**              int - 0 if success and non-zero otherwise
**
** Purpose:     Increment the running work items for the given trail.
**
**********************************************************
*/
int
IncrementRunningWI(int ID, int *wiCount, int *status)
{
    EDMRESETrailList *trl;
    int ret = 0;

    if (status == NULL)
    {
        return ret;
    }

    if (trl == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    *wiCount = trl -> IncrementRunningWIs();

    return 0;
}
```

```
/********************************************
**
**  Routine:    DecrementRunningWI
**
**
**  Inputs:     int ID - trail ID
**  Outputs:    int *wiCount - number of work items running after
**                             decrement.
**              int *status - status of the function if an error occured
**
**  Return Codes:
**              int - 0 if success and non-zero otherwise
**
**  Purpose:    Decrement the running work items for the given trail.
**
*********************************************/

int
DecrementRunningWI(int ID, int *wiCount, int *status)
{
    EDMRESTrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    if (wiCount == NULL)
    {
        *status = SCHED_BAD_PARAM;
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    *wiCount = trl -> DecrementRunningWIs();

    return 0;
}
```

```
/********************************************
**
**  Routine:    SetRunningWI
**
**  Inputs:     int ID - trail ID
**              int wiCount - number of work items to set.
**  Outputs:    int *status - status of the function if an error occured
**
**  Return Codes:
**              int - 0 if success and non-zero otherwise
**
**  Purpose:    Set the running work items for the given trail.
**
*********************************************/

int
SetRunningWI(int ID, int wiCount, int *status)
{
    EDMRESTrailList *trl;
    int  ret = 0;

    if (status == NULL)
    {
        return -1;
    }

    ret = LookupTrailObject(ID, &trl, status);

    if (ret != 0)
    {
        return ret;
    }

    if (trl == NULL)
    {
        return -1;
    }

    trl -> setRunningWIs(wiCount);

    return 0;
}
```

```
978
979    /****************************************************
980    **
981    ** Routine:   GetRunningWI
982    **
983    ** Inputs:    int ID - trail ID
984    **
985    ** Outputs:   int *wiCount - number of work items running;
986    **            int *status - status of the function if an error occured
987    **
988    ** Return Codes:
989    **            int - 0 if success and non-zero otherwise
990    **
991    ** Purpose:   Get the running work items for the given trail.
992    **
993    ****************************************************/
994
995
996
997  1 int
998  1 GetRunningWI(int ID, int *wiCount, int *status)
1000 1 {
1001 2     int ret = 0;
1002 2
1003 1     EDMWIETraillist *trl;
1005 1
1006 2     if (status == NULL)
1007 2     {
1008 2         return -1;
1009 1     }
1011 1
1013 1     if (wiCount == NULL)
1014 2     {
1015 2         *status = SCHED_BAD_PARAM;
1016 1         return -1;
1018 1     }
1019 2
1020 2     trl = LookupTrailObject(ID, &trl, status);
1021 1
1023 1     if (ret != 0)
1025 1     {
1026          return ret;
            }

            if (trl == NULL)
            {
                return -1;
            }

            *wiCount = trl -> getRunningWIs();

            return 0;
        }
```

```
1029
1030   /****************************************************
1031   **
1032   ** Routine:   PopWIFromTrailQueue
1033   **
1034   ** Inputs:    int ID - trail ID
1035   **
1036   ** Outputs:   int *status - status of the function if an error occured
1037   **            int *submitID - a place to put the submitID
1038   **            int *elementID - a place to put the element ID
1039   **
1040   ** Return Codes:
1041   **            int - 0 if success and non-zero otherwise
1042   **
1043   ** Purpose:   Gets the submit ID and element ID of the next work item
1044   **            to run.
1045   **
1046   ****************************************************/
1047
1048   int
1048   PopWIFromTrailQueue(
1049       int ID, int *submitID, int *elementID, int *status)
1050   {
1051       int ret = 0;
1052
1053       EDMWIETraillist *trl;
1054       EDMREScheduledWI *swi;
1056 1
1057 1     if (status == NULL)
1059 1     {
1060 2         return -1;
1061 2     }
1062 1
1063 1     if (submitID == NULL || elementID == NULL)
1065 1     {
1067 1         *status = SCHED_BAD_PARAM;
1068 2         return -1;
1069 2     }
1070 1
1071 2     ret = LookupTrailObject(ID, &trl, status);
1072 2
1074 1     if (ret != 0)
1075 2     {
1077 1         return ret;
1078 2     }
1079 2
1080 1     if (trl == NULL)
1081 1     {
1083 1         return -1;
1085 1     }
1086 1
1087 2     if (trl -> isTrailActive() == FALSE)
1088 2     {
1089 2         *status = SCHED_TRAIL_NOT_ACTIVE;
1090 1         return -1;
             }

             swi = trl > popScheduledWI();

             if (swi == NULL)
             {
                 *status = SCHED_NO_MORE_JOBS;
                 return -1;
         }
```

```
1089 1    }
     }

1091 1    *submitID = sw -> getSubmitObjectID();
1092 1    *elementID = sw -> getSubmitElementID();

1094 1    delete sw;

1096 1    return 0;
1097 1    }
```

```
1099      /**************************************************
          **
          **  Routine:     AddWIToTrailQueue
          **
          **  Inputs:      int ID - trail ID
          **               int submitID - the submitID of the work item
          **               int elementID - the element ID of the work item
          **
          **  Outputs:     int *status - status of the function if an error occured
          **
          **  Return Codes:  int - 0 if success and non-zero otherwise
          **
          **  Purpose:     Add the work item described by the submit ID and element
          **               ID to
          **               the specified trail queue.
          **
          **************************************************
1113      */
1114      int
1115      AddWIToTrailQueue(int ID, int submitID, int elementID, int *status)
1116    1 {
1118    1    int ret = 0;
1119
1120    1    if (status == NULL)
1121    2    {
1123    1       return -1;
1124    2    }
1125    1
1126    1    ret = NewSchedWI(ID, submitID, elementID, status);
1128    1
1129    1    if (ret <= 0)
1130    2    {
1131    2       return -1;
1132    1    }
1133    1
1135    1    return 0;
1136    1 }
```

```
1138
1139   /* *****************************************************************
1140   **
1141   ** Routine:    FindTrailQueueOfWI
1142   **
1143   ** Inputs:     int handle - handle to identify work item
1144   **             int *status - status of the function if an error occured
1145   **             int *ID - a place to put the trail ID
1146   **
1147   ** Return Codes:
1148   **             int  - 0 if success and non-zero otherwise
1149   **
1150   ** Purpose:    Gets the trail ID of the work item.
1151   **
1152   ** *****************************************************************
1153   */
1154   int
1155   FindTrailQueueOfWI(int handle, int *ID, int *status)
1156   {
1157      EDMRESubmitElement *se;
1158      int    soID;
1159      int    seID;
1160      int    ret;
1161      int    findArg;
1162
1163
1164      if (status == NULL)
1165      {
1166         return -1;
1167      }
1168
1169      if (ID == NULL)
1170      {
1171         *status = SCHED_BAD_PARAM;
1172         return -1;
1173      }
1174
1175      ret = getSubmitIDs(handle, &soID, &seID, status);
1176
1177      if (ret != 0)
1178         return ret;
1179
1180      ret = LookupSubmitElement(soID, seID, &se, status);
1181
1182      if (ret != 0)
1183         return ret;
1184
1185      fa.alternate = se -> IsAlternateTrailset();
1186      se -> getTemplate(fa.templatename, TEMPLNAME, SIZE8);
1187      fa.trailnum = 0;
1188
1189      LockScheduleMutex();
1190
1191      trail.lists.apply(findTrail, &fa);
1192
1193      UnlockScheduleMutex();
1194
1195      if (fa.trailnum == 0)
1196         return -1;
1197
1198      *ID = fa.trailnum;
1199
1200      return 0;
```

```
1201   }
```